BRIDGING
JORAM 4.0 AND XMQ

Author: Frédéric Maistre (Bull SA)
February 13, 2004

TABLE

1. Introduction 3

2. Concepts and implementation 5

2.1. Messages exchange 6

2.2. Acknowledgement policy 6

2.3. Message selection 8

2.4. Connection failure handling 9

3. User manual 10

3.1. Configuring the foreign platform 10

3.2. Configuring JORAM 10

3.3. Steps 11

3.4. Failures 12

4. WebSphere MQ example 13

4.1. Configuring and starting JORAM 13

4.2. WebSphere MQ setup 14

4.3. Administering JORAM 14

4.4. The JORAM subscriber 15

4.5. The WebSphere MQ sender 16

4.6. As a conclusion… 17

1. Introduction

The JORAM 4.0 release introduces a bridge feature. This bridge allows a JORAM client application to communicate with a JMS destination hosted by a foreign JMS server (let's call it *XMQ*) in a completely standard way.

The link between the JORAM and *XMQ* heterogeneous platforms is provided by a specifically configured JORAM destination (a queue or a topic), connected to a *XMQ* destination as a standard (JMS 1.0.2b or JMS 1.1) client application (as illustrated by picture 1).

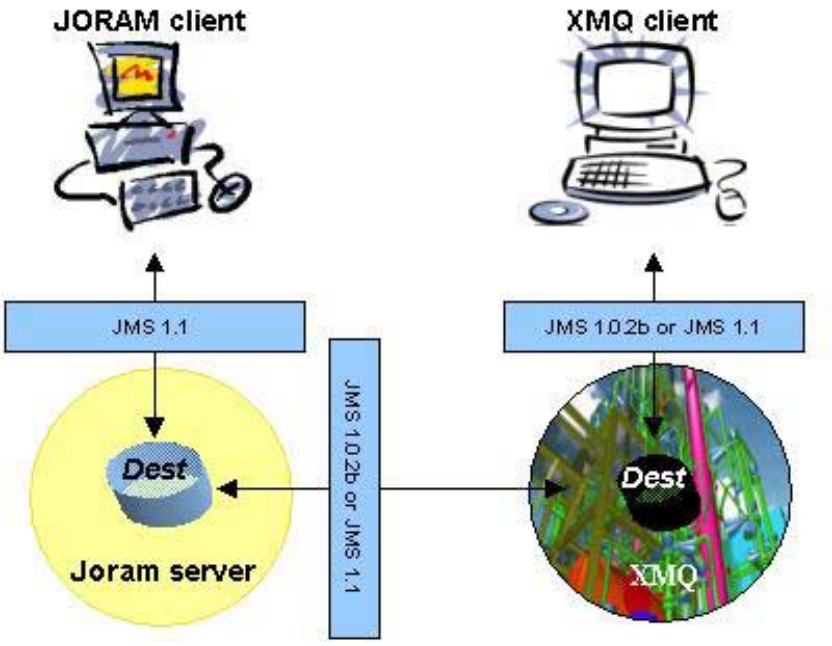

Figure 1: a JORAM client communicating with a XMQ client

When the JORAM client sends a message on the JORAM bridge destination, the message is actually forwarded to the *XMQ* destination. If the JORAM client requests a message from the JORAM bridge destination, the request is forwarded to the *XMQ* destination. The *XMQ* destination answers by returning a JMS message, which is stored in the JORAM bridge destination and eventually consumed by the JORAM client.

From the JORAM client perspective, the target destination is a JORAM destination accessed through the JMS interfaces. It is a fully standard client. The facts that the messages it produces finally reach the *XMQ* destination, and that the messages it consumes originally come from the *XMQ* destination, are totally transparent to the JORAM user.

This document presents how the JORAM bridge feature works, how to use it, and finally shows sample codes for bridging JORAM with IBM's WebSphere MQ platform.

2. Concepts and implementation

As said in the introduction, the JORAM JMS bridge feature is provided by specialized JORAM destinations. The communication between a JORAM client and a JORAM bridge destination is standard. It is the processing of the client requests within the bridge destination that is specialized.

A bridge destination has 3 main roles:

- as a JORAM destination, it receives messages sent by JORAM JMS producers, and requests sent by JORAM JMS consumers;
- as a *XMQ* JMS producer, it forwards the received messages to the *XMQ* destination; as a *XMQ* JMS consumer, it requests messages from the *XMQ* destination;
- as a *XMQ* JMS consumer, it consumes messages coming from the *XMQ* destination; as a JORAM destination, it delivers those messages to the JORAM JMS consumers.

The JMS resources involved in the JORAM client – *XMQ* client communication are illustrated by the next picture.

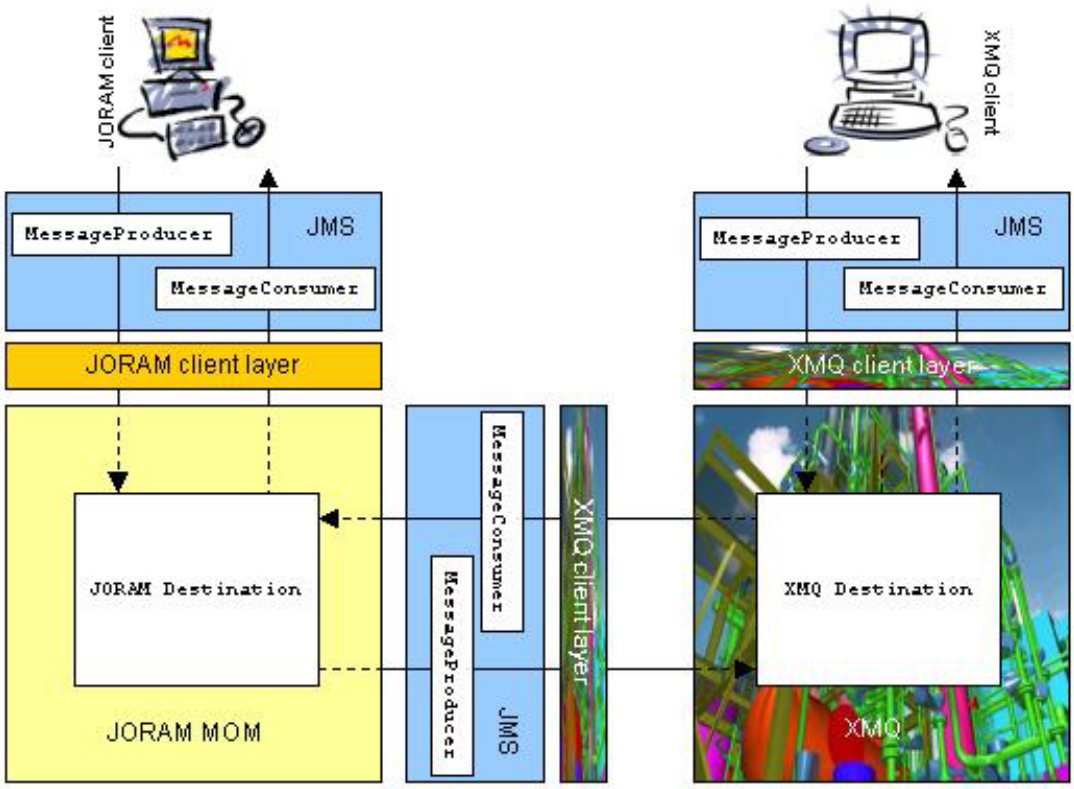

Figure 2: bridge communication diagram

2.1. Messages exchange

Communication modes

The JORAM bridge destination may either be a queue or a topic. And in both cases it may either communicate with a foreign queue or a foreign topic. A JORAM client application may then interact with a *XMQ* destination normally incompatible with the communication mode. For example, a JORAM client can interact with a *XMQ* queue in a Publish/Subscribe way.

Message sending

A message sent by a JORAM client application first reaches the bridge destination. It is then forwarded to the *XMQ* destination. Once successfully delivered to the *XMQ* platform, it is removed from the bridge destination. While the message was hosted by the JORAM bridge destination (between its arrival and its delivery to the foreign platform), it has never been available for delivery to a JORAM client. Even if the bridge destination had pending consumers requests, they wouldn't have been answered with this message. Messages arriving on a bridge destination and coming from a JORAM client application are only deliverable to a foreign JMS destination.

Message consumption

A consuming request sent by a JORAM client application first reaches the bridge destination. It is then forwarded to the *XMQ* destination. When the *XMQ* target destination is a topic, the JORAM bridge destination, when receiving the first consuming request, durably subscribes to the *XMQ* topic. This for not loosing messages if the JMS connection between the bridge destination and the *XMQ* platform fails (connection failures handling is addressed section 2.4). Messages delivered by the XMQ platform are consumed by the JORAM bridge destination. At that point they are converted into JORAM JMS messages (if the conversion fails, the delivery is rolled back, see section 2.2). Finally, those messages are consumed by the JORAM client applications.

Queue browsing

The JMS API allows a client application to browse the messages available for delivery on a queue. When requesting to browse a JORAM queue bridge, the "browse" request is forwarded to the *XMQ* destination if it is a queue. It eventually returns an enumeration of messages, converted into JORAM JMS messages. The enumeration is finally forwarded to the original requester. If the bridge destination is connected to a *XMQ topic*, the request is directly answered by an empty enumeration.

2.2. Acknowledgement policy

Acknowledgements are managed between the JORAM client and the JORAM bridge destination, and between the JORAM bridge destination and the *XMQ* destination, independently.

The picture 3 shows a first scenario of message rollback. In fact, it is the only possible case of message denying between the JORAM bridge destination and the *XMQ* platform. It occurs when the JMS message delivered by the *XMQ* destination appears not to be readable and thus can not be converted into a JORAM JMS message.

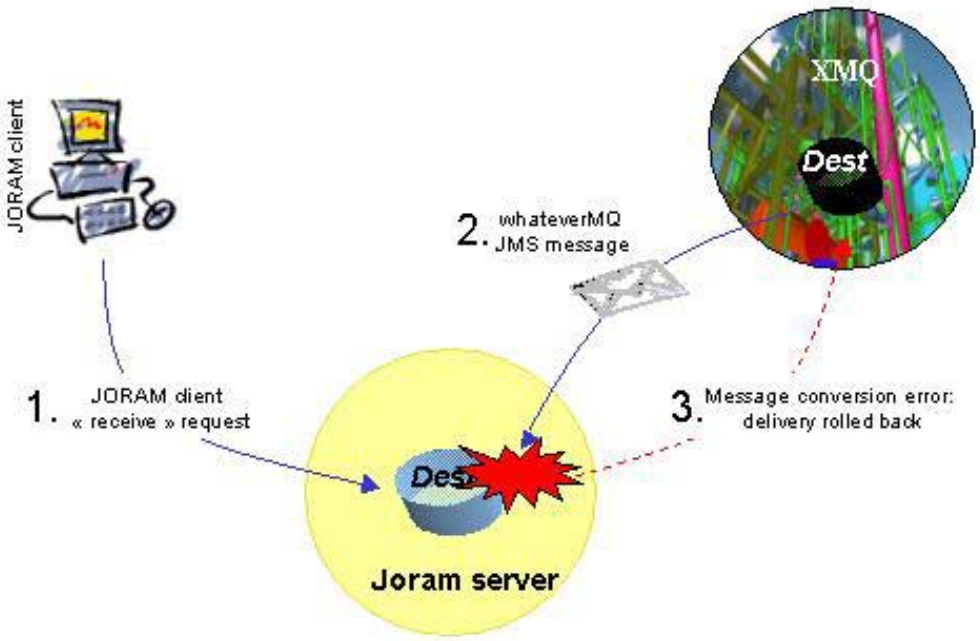

Figure 3: delivery rolled back by the JORAM bridge destination

This situation will likely evolve towards the “poison” message scenario, where a JMS client rolls back its session each time it receives the redelivered failing message. In order to avoid this, it is hoped that *XMQ* provides a way to log such messages into a dead message queue. If yes, *XMQ* should be configured for doing so.

Once a message delivered by the *XMQ* destination has been successfully converted into a JORAM JMS message, the delivery is acknowledged. *XMQ* does not hold the message anymore, it is JORAM which is now responsible for safely distributing it.

The picture 4 shows the second possible scenario involving a rollback or message denying. It occurs when the JORAM client application, which finally consumes the message, either fails before acknowledging the message or explicitly rolls back its session.

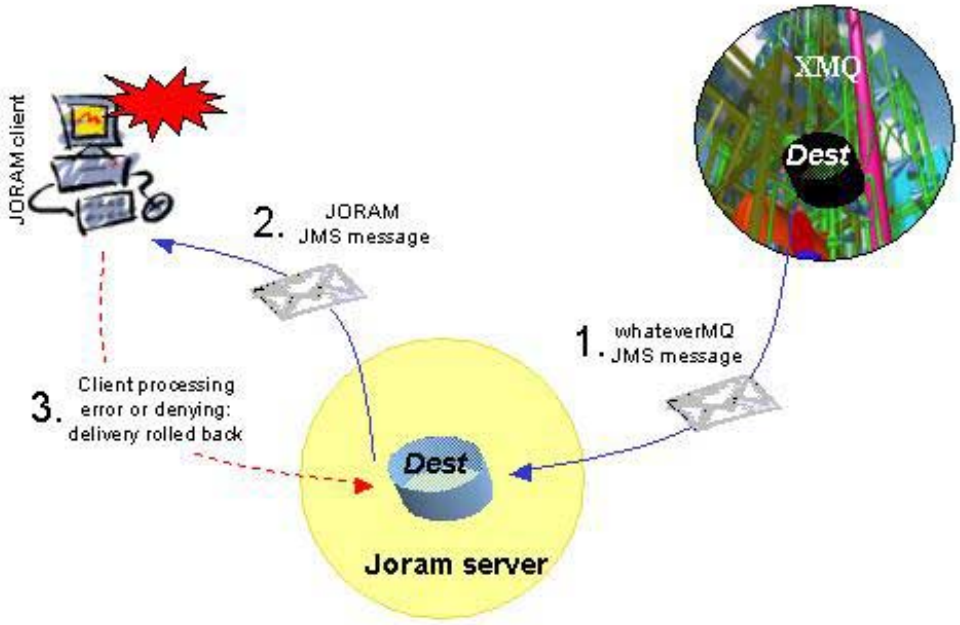

Figure 4: delivery rolled back by the JORAM client application

The consequence of this message denying is the same as between any JORAM client and JORAM destination. The message is available again for delivery, or logged to a dead message queue if any. In all cases the message stays on the JORAM platform, the *XMQ* destination is not notified of the JORAM client acknowledgements or denials.

2.3. Message selection

As for acknowledgements, message selection is handled separately between a JORAM client and the JORAM bridge destination it interacts with, and between the JORAM bridge destination and the *XMQ* destination. Selectors set-up is done at different times. The selector used for filtering the messages on the *XMQ* destination is set at administration time, when configuring the JORAM bridge destination. It can not be changed once the bridge destination has been created. The selector used by the JORAM client is set as a standard selector, when creating the `MessageConsumer` instance.

The picture 5 shows how those selectors operate. The bridge's selector selects messages with a given property value above 2. As a consequence, only the messages which property value is above 2 will actually be transferred from *XMQ* to JORAM.

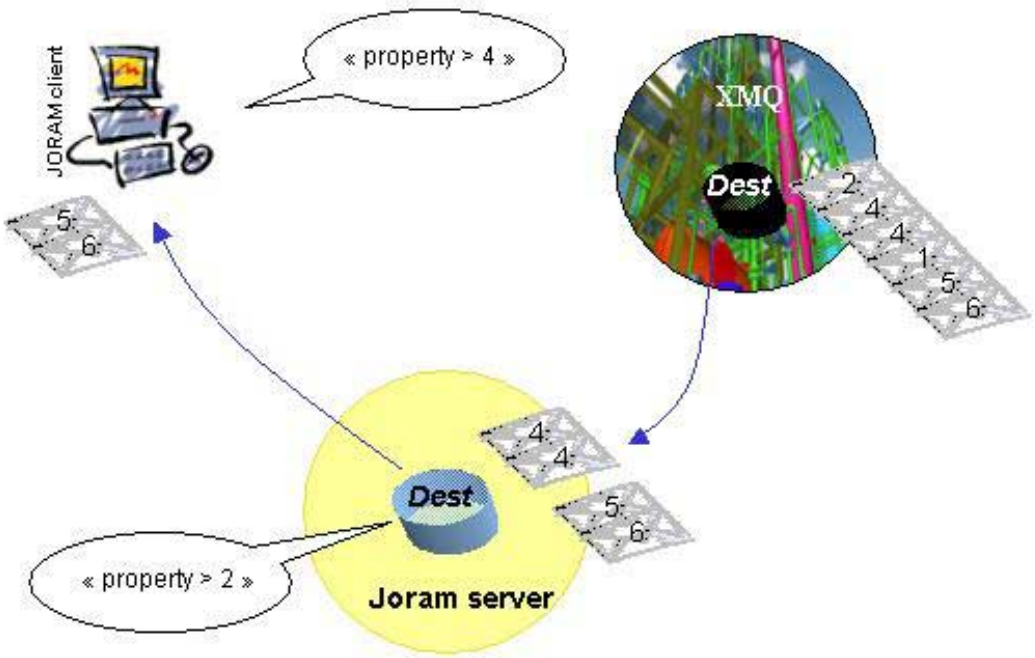

Figure 5: how selectors operate

The JORAM client application selects messages with the same property value above 4. It then consumes the corresponding messages available on the bridge destination. Other clients may consume the other messages by setting different selectors.

2.4. Connection failure handling

From an architectural point of view, the *XMQ* server might be seen as a JORAM server of a JORAM distributed configuration. It might happen that the JMS connection between the JORAM bridge destination and the *XMQ* platform breaks. This case is processed as any connection failure case between two JORAM servers. An automatic reconnection process is launched when the failure is detected (through the setting of a `javax.jms.ExceptionListener` by the JORAM bridge destination). When the JORAM bridge destination finally reconnects, the pending messages or requests are re-routed to the *XMQ* platform.

As a consequence, disconnections between JORAM and *XMQ* are totally transparent to the user, as disconnections between JORAM servers of a distributed JORAM platform.

The reconnection process is as follows:

1. first step: 30 connection trials, one per second;
2. second step: 55 connection trials, one every 5 seconds;
3. last step: infinite connection trials, one every minute.

3. User manual

A JORAM bridge client is a fully standard JMS client which has no visibility of the foreign *XMQ* platform. Of course achieving this requires to properly configure and administer the JORAM and *XMQ* platforms.

3.1. Configuring the foreign platform

Prerequisites

The prerequisites on the foreign JMS server to bridge with JORAM are as follows:

- full implementation of the JMS 1.0.2b or JMS 1.1 API;
- particularly, the `javax.jms.ExceptionListener` must be supported.

Configuration and administration

The foreign *XMQ* platform should be appropriately configured for hosting a destination, and authorizing the connection of an external client.

JMS administered objects (`javax.jms.ConnectionFactory` and `javax.jms.Destination`) that will be used by the JORAM bridge to connect to the *XMQ* platform and access the right destination should be created and bound to a JNDI compliant naming server.

3.2. Configuring JORAM

Configuration

JORAM's server classpath must provide access to the following libraries:

- all *XMQ* client jars;
- a `jms.jar` library compatible with the *XMQ* JMS implementation;
- the client jars of the used JNDI server;
- a `jndi.jar` library compatible with the naming server used for binding the *XMQ* administered objects.

Administration

Creating a JORAM bridge destination is done through the `org.objectweb.joram.client.jms.Queue.create` and `org.objectweb.joram.client.jms.Topic.create` static methods.

The parameters of these methods are:

- `int serverId`: identifier of the server on which deploying the destination;
- `String className`: name of the bridge's class:
  - "org.objectweb.joram.mom.dest.BridgeQueue" for a queue,
  - "org.objectweb.joram.mom.dest.BridgeTopic" for a topic;

- `java.util.Properties prop`: properties configuring the bridge destination.

The following properties are required for setting the bridge destination:

- “connectionFactoryName”: JNDI name used to bound the *XMQ* JMS `ConnectionFactory` object;
- “destinationName”: JNDI name used to bound the *XMQ* JMS `Destination` object;
- “jmsMode”: says what is the type of the *XMQ* `ConnectionFactory` instance bound to JNDI:
  - “ptp” for a `QueueConnectionFactory`,
  - “pubsub” for a `TopicConnectionFactory`,
  - “unified” for a `ConnectionFactory`.

The following properties are optional:

- “userName”: user identification that should be used by the bridge destination for opening a connection to *XMQ*; if not provided, the connection will be opened with no identification.
- “password”: user password that should be used by the bridge destination for opening a connection to *XMQ*; if not provided, the connection will be opened with no identification.
- “jndiFactory”: name of the JNDI initial context factory class (for example “fr.dyade.aaa.jndi2.client.NamingContextFactory”); if not provided, it is expected that environment variables are set or that a `jndi.properties` file is accessible through the server’s classpath.
- “jndiUrl”: URL of the JNDI server (for example “joram://localhost:16400”); if not provided, it is expected that environment variables are set or that a `jndi.properties` file is accessible through the server’s classpath.
- “clientId”: provided if *XMQ* requires the setting of such an identifier on its client connection.
- “selector”: selector expression used for filtering messages on the *XMQ* destination.

3.3. Steps

In order to be able to bind the foreign JMS provider administered objects, a naming server is the first thing to start. And in order to be able to successfully deploy a JORAM bridge destination, the foreign JMS administered objects must have been bound. As a consequence, the start-up steps are as follows:

1. starting a JNDI server;
2. creating the foreign JMS administered objects, binding them to the JNDI server;
3. starting and administering JORAM.

If the JNDI server used is hosted by the JORAM server, then the JORAM server is the first thing to start (in order to start the JNDI service). Then the XMQ JMS administered objects should be created and bound. Finally, the JORAM server should be administered, and particularly the bridge destination should be created.

3.4. Failures

If one of the mandatory configuration properties is missing, the bridge destination creation will fail (the static `create` method will throw an `AdminException`).

If the provided properties do not allow to successfully set a link with the foreign JMS server, the bridge won't be usable. But as the initialisation process is asynchronous, the administrator won't get any information about this failure.

Simply, when sending or requesting messages to/from this bridge, ERROR messages will be logged. The possible error cases are:

- JNDI server not reachable (due to invalid "jndiFactory" or "jndiUrl" values);
- the administered objects could not be retrieved from the JNDI server (invalid "cnxFactoryName" or "destinationName" values);
- missing *XMQ* client libraries in JORAM server's classpath, preventing the successful re-construction of the retrieved administered objects;
- communication mode ("jmsMode" value) not compatible with the retrieved administered objects (`TopicConnectionFactory` instance for a "ptp" mode for example);
- invalid retrieved administered objects types (for example a `QueueConnectionFactory` instance and a `Topic` destination);
- incorrect JMS user identification ("userName and "password" values).

4. WebSphere MQ example

This example links JORAM with IBM's WebSphere MQ platform (formerly MQ Series). It shows a JORAM topic bridge linked to a WebSphere MQ queue.

4.1. Configuring and starting JORAM

Configuring JORAM

We will consider a simple JORAM platform made of one standard server (hosting a JNDI service). It is described by the following `a3servers.xml` file:

```
<?xml version="1.0"?>
<config>
  <server id="0" name="S0" hostname="localhost">
    <service
     class="org.objectweb.joram.mom.proxies.ConnectionManager"
     args="root root"/>
    <service
     class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
     args="16010"/>
    <service
     class="fr.dyade.aaa.jndi2.server.JndiServer"
     args="16400"/>
  </server>
</config>
```

The following WebSphere MQ libraries must be added to JORAM's server `lib/` directory:

- `com.ibm.mqjms.jar`
- `com.ibm.mqbind.jar`
- `com.ibm.mq.jar`
- `connector.jar`
- `fscontext.jar`

The JORAM distribution already provides the needed `jms.jar`, `jndi.jar` and `jndi.properties`. No need to add those in the `lib/` directory.

Starting JORAM

Starting JORAM is required first because the JORAM server provides the JNDI service needed for binding WebSphereMQ's administered objects.

Launching the JORAM server is done as usual:

```
java –DTransaction=fr.dyade.aaa.util.Atransaction
     fr.dyade.aaa.agent.AgentServer 0 ./s0
```

4.2. WebSphere MQ setup

The WebSphere MQ administration tool should be used for creating a queue. Let's call this queue "foreignQueue".

JMS administered objects should then be created. They will eventually be bound to JORAM's JNDI. For that, WebSphere MQ `JMSAdmin.config` file must be edited as follows:

```
INITIAL_CONTEXT_FACTORY=
  fr.dyade.aaa.jndi2.client.NamingContextFactory
PROVIDER_URL=joram://localhost:16400
```

The JMSAdmin command should then be launched and the JMS objects created and bound to JNDI (with names "foreignCF" and "foreignDest"):

```
InitCtx> DEFINE QCF(foreignCF)
InitCtx> DEFINE Q(foreignDest) QUEUE(foreignQueue)
```

4.3. Administering JORAM

Administering JORAM consists in creating and configuring the required bridge destination, and authorizing a client access to it. The following code is proposed:

```
public class AdminSample
{
  public static void main(String[] args) throws Exception
  {
    AdminModule.connect("root", "root", 60);

    Properties prop = new Properties();
    // Setting the communication mode to PTP for interacting
    // with WebSphere MQ
    prop.setProperty("jmsMode", "ptp");

    // Setting the JNDI names of the WebSphere MQ
    // administered objects
    prop.setProperty("connectionFactoryName", "foreignCF");
    prop.setProperty("destinationName", "foreignDest");

    // Creating the bridge topic on JORAM server 0
    String className = "org.objectweb.joram.mom.dest.BridgeTopic";
    Topic bridgeTopic = Topic.create(0, className, prop)

    // Providing free access to it
    bridgeTopic.setFreeReading();
    bridgeTopic.setFreeWriting();
```

```
      // Creating an anonymous user access on JORAM server 0
      javax.jms.ConnectionFactory cf =
        TcpConnectionFactory.create("localhost", 16010);
      User user = User.create("anonymous", "anonymous", 0);

      // Binding the JORAM administered objects
      javax.naming.Context jndiCtx =
        new javax.naming.InitialContext();
      jndiCtx.bind("bridgeDest", bridgeDest);
      jndiCtx.bind("cf", cf);

      jndiCtx.close();

      AdminModule.disconnect();
    }
  }
```

4.4. The JORAM subscriber

The development of the JORAM subscriber is fully standard. The JORAM application is apparently in relationship with a pure JORAM topic destination, but in fact it consumes messages coming from the WebSphere MQ queue destination, in a JMS 1.1 way, through the Publish/Subscribe communication mode.

The following code is proposed:

```
  public class ConsumerSample
  {
    public static void main(String[] args) throws Exception
    {
      // Retrieving the JORAM administered objects
      javax.naming.Context jndiCtx =
        new javax.naming.InitialContext();
      Destination dest = (Destination) jndiCtx.lookup("bridgeDest");
      ConnectionFactory cnxFact =
       (ConnectionFactory) jndiCtx.lookup("cf");
      jndiCtx.close();

      Connection cnx = cnxFact.createConnection();
      Session sess =
        cnx.createSession(false, Session.AUTO_ACKNOWLEDGE);
      MessageConsumer cons = sess.createConsumer(dest);
      cons.setMessageListener(new MsgListener());

      //…
```

```
      // …

      cnx.start();
      System.in.read();
      cnx.close();
    }
  }
```

4.5. The WebSphere MQ sender

The WebSphere MQ sender is a JMS 1.0.2b standard sender, sending messages to the WebSphere MQ queue through a `QueueSender` resource.

The following code is proposed:

```
  public class SenderSample
  {
    public static void main(String[] args) throws Exception
    {
      // Retrieving the WebSphere MQ administered objects
      javax.naming.Context jndiCtx =
        new javax.naming.InitialContext();
      Queue queue = (Queue) jndiCtx.lookup("foreignDest");
      QueueConnectionFactory cnxFact =
       (QueueConnectionFactory) jndiCtx.lookup("foreignCF");
      jndiCtx.close();

      QueueConnection cnx = cnxFact.createQueueConnection();
      QueueSession sess = cnx.createQueueSession(true, 0);
      QueueSender sender = sess.createSender(queue);
      TextMessage msg = sess.createTextMessage();

      // Sending 10 messages to the queue
      for (int i = 1; i < 11; i++) {
        msg.setText("WebSphere MQ message " + i);
        sender.send(msg);
      }
      // Commiting the session
      sess.commit();

      cnx.close();
    }
  }
```

4.6. As a conclusion...

The WebSphere MQ demo distribution is JMS 1.0.2b compliant and does not provide the Publish/Subscribe communication mode. This example illustrates JORAM as a JMS 1.1 Publish/Subscribe broker to WebSphere MQ.